TD n°11 – Corrigé

Exercice 1 *Applications des piles*

Dans cet exercice on suppose écrite en C un type structure `pile`, représentant une pile d'entiers. On dispose de fonctions `pile* pile_vide()`, `bool est_vide_pile(pile* p)` `int depile(pile* p)`, `void empile(pile* p, int e)` et `int longueur(pile* p)` qui fonctionnent avec effets de bord.

1. Écrire une fonction qui échange les deux premiers éléments d'une pile.

```
void echange(pile* p){
   int a = depile(p);
   int b = depile(p);
   empile(a, p);
   empile(b,p);
}
```

2. Écrire une fonction qui dépile et affiche le n-ième élément. À quoi faut-il penser en terme de programmation défensive ?

```
void nieme(pile* p, int n){
   int i = 1;
   pile* pbis = pile_vide();
   while (i<n){
       assert(not est_vide_pile(p)); //On vérifie qu'il y a bien plus de i éléments dans la pile
       empile(pbis, depile(p));
       i+=1;
   }
   //On a enlevé n-1 éléments a la pile, donc on affiche le prochain
   printf("%d\n", depile(p));

   //Maintenant il faut rempiler tout les éléments qui étaient avant le nieme
   while (not est_vide_pile(pbis)){
       empile(p, depile(pbis));
   }
}
```

3. Écrire une fonction qui prend en entrée un pile et modifie la pile pour que le premier élément devienne le dernier, mais les autres restent dans le même ordre.

```
void premierdernier(pile* p){
   pile* pbis = pile_vide();
   while (not est_vide_pile(p)){
       int el = depile(p)
       empile(pbis, el);
   }
   //On a retiré tous les éléments de la pile. el vaut la valeur du dernier élément
   //On rempile donc tous les éléments, sauf el (qui est premier dans pbis)
   depile(pbis);
   while (not est_vide_pile(pbis)){
       empile(p, depile(pbis));
   }
   //Enfin on rempile el
   empile(p, el);
}
```

4. Écrire une fonction qui prend en entrée une pile et renverse l'ordre des éléments dans la pile
   Ici on a à nouveau besoin de vider toute la pile, mais pas dans une pile, qui ne permet pas d'accéder librement aux éléments. On va plutôt utiliser un tableau.

```
void renverse(pile* p){
   int l = longueur(p);
   int* sto = malloc(l*sizeof(int));
   int i=0;
   while (not est_vide_pile(p)){
       sto[i] = depile(p);
       i+=1;
   }
   i-=1; //i est désormais l'indice de la derniere case du tableau
   //On a retiré tous les éléments de la pile. On va les rajouter mais a l'envers
   while (i!=-1){
```

```
        empile(p, tab[i]);
    }
}
```

5. Écrire une fonction qui prend en entrée une pile et la découpe en deux. Une des deux parties sera renvoyée dans une autre pile et l'autre sera mise dans la pile d'origine (pour faire cela il faut un pointeur)
   Pour découper en deux le plus simple est de calculer la longueur n et dépiler les $n/2$ premiers éléments dans une autre pile. Pour préserver l'ordre, on retourne celle-ci avec la fonction de la question précédente.

```
void coupe(pile* p){
    pile* pbis = pile_vide();
    while (not est_vide_pile(p)){
        empile(pbis, depile(p));
    }
    renverse(pbis);
}
```

6. Écrire une fonction qui prend en entrée deux piles et les "mélange" aléatoirement. On suppose qu'on dispose d'une fonction `int zero_ou_un()` qui renvoie aléatoirement 0 ou 1 avec probabilités égales.
   Tant que les deux piles ne sont pas vides, on tire au hasard dans laquelle des deux on dépile.

```
void melange(pile* p1, pile* p2){
    pile* pmelange = pile_vide();
    while (not est_vide_pile(p1) && not est_vide_pile(p2)){
        int tire = zero_ou_un();
        if (tire == 1){empile(pmelange, depile(p1);}
        else {empile(pmelange, depile(p2);}
    }
    //On s'occupe de ce qui reste
    while (not est_vide_pile(p1)){
        empile(pmelange, depile(p1);
    }
    while (not est_vide_pile(p2)){
        empile(pmelange, depile(p2);
    }
}
```

Exercice 2 *Nombres de Hamming et file*

Les nombres de Hamming sont les nombres de la forme $2^a3^b5^c$ pour a, b, c entiers naturels quelconques. Les premiers entiers de Hamming sont 1,2,3,4,5,6,6,9,10,12,15,16,18,20,...

Le but de cet exercice est de générer la liste des n premiers nombres de Hamming. L'apporche naïve consiste à parcourir les entiers en vérifiant s'ils sont des nombres de Hamming, jusqu'à en avoir trouvé n.

1. Écrire une fonction `est_hamming : int->bool` qui vérifie si un entier est un nombre de Hamming. On doit vérifier si les facteurs premiers de e sont uniquement 2,3 et 5. On divise donc e par ces nombres tant que possible et on vérifie si le résultat est bien 1.

```
let est_hamming e =
  let m = ref e in
  while !m mod 2 = 0 do m:=!m/2 done;
  while !m mod 3 = 0 do m:=!m/3 done;
  while !m mod 5 = 0 do m:=!m/5 done;
  if !m = 1 then true
  else false;;
```

2. Écrire une fonction `hamming_naif : int->int list` qui prend en entrée n est renvoie la liste des n premiers nombres de Hamming.

```
let hamming_naif n =
  let rec aux n m = match n with (*n est le nombre de nombres de Hamming restants vdash a
trouver. m est le candidat actuel*)
      |0 -> []
      |_ -> if est_hamming m then m::aux (n-1) (m+1) (*Si on a trouvé un nombre de hamming,
              il en reste n-1 vdash a trouver et le candidat devient m+1*)
            else aux n (m+1) (*Sinon n ne change pas et le candidat devient m+1*)
  in aux n 1;;
```

Si cette approche fonctionne bien pour les premiers termes, plus n grandit et plus les nombres de Hamming sont éloignés les uns des autres (par exemple le 1999e est 8 100 000 000 et le 2000e 8 153 726 976). Il devient donc trop coûteux d'explorer tous les entiers pour trouver les nombres de Hamming.

On va plutôt générer les nombres de Hamming à partir d'autres nombres de Hamming. On utilise pour ce faire trois files f_2, f_3 et f_5, qui initialement contiennent le nombre 1 et on leur applique l'algorithme suivant, jusqu'à avoir affiché n valeurs :

- on détermine le plus petit élément entre les trois têtes de files, noté k, et on l'affiche.
- on retire k des files où il est présent.
- on enfile sur la file f_2 l'entier $2k$, sur f_3 l'entier $3k$ et sur f_5 l'entier $5k$.

Cet algorithme repose sur le fait que tout nombre de Hamming est le produit par 2,3 ou 5 d'un autre nombre de Hamming plus petit.

Pour l'implémentation on utilisera le module Queue de Ocaml, qui propose une implémentation mutable de file. Les primitives ont les noms suivants (en anglais), le type `'a t` désigne une file :

- `Queue.create : unit -> 'a t` qui crée une file vide
- `Queue.push : 'a -> 'a t -> unit` qui ajoute un élément
- `Queue.pop : 'a t -> 'a` qui retire et renvoie l'élément le plus ancien
- `Queue.peek : 'a t -> 'a` qui renvoie sans retire l'élément le plus ancien

Les fonctions `peek` et `pop` lèvent l'exception `Empty` si la file est vide.

3. Traduire l'algorithme en Ocaml.

```
let hamming n =
  let f2 = Queue.create() and f3= Queue.create() and f5 = Queue.create() in
  Queue.push 1 f2 ; Queue.push 1 f3 ; Queue.push 1 f5 ;
  for i = 1 to n do
      (*On regarde les 3 têtes et on compare*)
      let x2 = Queue.peek f2 and x3 = Queue.peek f3 and x5 = Queue.peek f5 in
      let x = min x2 (min x3 x5) in

      (*Affichage*)
      print_int x ; print_char ' ' ;

      (*x peut être dans plusieurs files vdans plusieurs fois*)
      if x = x2 then ( let _ = Queue.pop f2 in () );
      if x = x3 then ( let _ = Queue.pop f3 in () );
      if x = x5 then ( let _ = Queue.pop f5 in () );

      (*On ajoute les nouveaux éléments*)
      Queue.push (2*x) f2 ; Queue.push (3*x) f3 ; Queue.push (5*x) f5 ;
  done ;;
```

4. (*) L'inconvénient de la démarche précédente est que le même nombre peut se retrouver dans plusieurs des trois files. Modifier votre fonction pour que cela ne soit plus le cas.

   Si x est multiple de 3 et 5 alors il se retrouvera dans f_3 et f_5. On peut remarquer qu'il se retrouvera d'abord dans f_5 car $x/5 < x/3$. Une solution est de n'ajouter un nombre à f_3 que s'il n'est pas multiple de 5. De la même manière, on n'ajoutera un nombre à f_2 que si ce n'est ni un multiple de 3, ni un multiple de 5.

   Dans cette nouvelle version, f_2 sert à générer les puissances de 2, f_3 sert à générer les nombres de la forme 2^a3^b avec $b \neq 0$ et f_5 sert à générer les nombres de la forme $2^a3^b5^c$ avec $c \neq 0$.

```
let hamming_bis n =
 let f2 = Queue.create() and f3= Queue.create() and f5 = Queue.create() in
 Queue.push 1 f2 ; Queue.push 1 f3 ; Queue.push 1 f5 ;
 for i = 1 to n do
     (*On regarde les 3 têtes et on compare*)
     let x2 = Queue.peek f2 and x3 = Queue.peek f3 and x5 = Queue.peek f5 in
     let x = min x2 (min x3 x5) in

     (*Affichage*)
     print_int x ; print_char ' ' ;

     (*x peut être dans plusieurs files vdans plusieurs fois*)
     if x = x2 then ( let _ = Queue.pop f2 in () );
     if x = x3 then ( let _ = Queue.pop f3 in () );
     if x = x5 then ( let _ = Queue.pop f5 in () );
```

```
        (*On ajoute les nouveaux éléments*)
        Queue.push (5*x) f5 ;
        if x mod 5 <> 0 then begin
            Queue.push (3*x) f3;
            if x mod 3 <> 0 then Queue.push (2*x) f2
        end;
    done ;;
```

Exercice 3 *Permutations et piles*

Une permutation de $[|1,n|]$ est une manière de réarranger les entiers de 1 à n. Par exemple pour $n = 5$, (1 2 4 5 3) est une permutation. (1 2 3 4 5) en est une aussi.

on peut aussi le voir comme les 5-uplets dont les éléments sont exactement ceux de $[|1,n|]$, sans répétitions.

On dit qu'une permutation (a_1 a_2 ... a_n) de $[|1,n|]$ peut être engendrée par une pile lorsqu'il est possible, à partir de la permutation (1 2 ... n) et d'une pile (initialement vide), d'afficher la séquence de sortie (a_1 a_2 ... a_n) en utilisant uniquement les opérations suivantes :

- empiler l'élément suivant dans la permutation d'entrée.
- dépiler un élément de la pile et l'afficher

Par exemple, si E et D désignent respectivement les deux opérations permises, la permutation (2 3 1) est engendrée par la suite d'opérations EEDEDD.

1. Parmi les permutations suivantes, lesquelles peuvent être engendrées par une pile ?

   (3 1 2), (3 4 2 1), (4 5 3 7 2 1 6), (3 5 7 6 8 4 9 2 10 1)

   (3 1 2) ne peut pas être engendrée, pour afficher 3 en premier il faut empiler 1, 2 et 3 mais alors 1 est coincé en dessous de 2 dans la pile.
   EEEDEDDD permet d'engendrer (3 4 2 1).
   (4 5 3 7 2 1 6) ne peut pas être engendrée.
   EEEDEEDEEDDEDDEDDEDD permet d'engendrer (3 5 7 6 8 4 9 2 10 1).
2. Montrer que s'il existe un triplet $(i,j,k) \in [|1,n|]^3$ tel que $i < j < k$ et $a_j < a_k < a_i$, alors la permutation (a_1 a_2 ... a_n) ne peut pas être engendrée par une pile.
   Supposons que (a_1 a_2 ... a_n) puisse être engendré par une pile.
   Comme les a_i sont donnés en entrée selon l'ordre de leurs valeurs, a_j est empilé avant a_k qui est empilé avant a_i.
   Donc au moment de dépiler a_i, a_k et a_j sont dans la pile et a_k est plus haut dans l'ordre de priorité que a_j (il est plus proche de la sortie si on représente séquentiellement). Pour pouvoir dépiler a_j, il est nécessaire que a_k soit dépilé avant, ce qui contredit que $j < k$.
3. Écrire une fonction Caml `est_engendrable : int list -> bool` déterminant si une permutation peut être engendrée par une pile. Dans le cas d'une réponse positive, la fonction affichera la suite d'opérations permettant de la produire. Les permutations seront représentées par le type `int list`.
   Nous allons utiliser un accumulateur qui va mémoriser la valeur i du plus grand entier a avoir été empilé. Lorsqu'il va falloir dépiler l'entier j, nous allons commencer par empiler les entiers compris entre $i+1$ et j (si $i < j$) puis dépiler j s'il se trouve au sommet de la pile. Dans le cas contraire, c'est que la permutation n'est pas engendrable.

   ```
   let engendrable =
     let p = creer() in
     let rec aux i = function
         | [] -> true
         | j::q -> for k = i+1 to j do empile k p ; print_char 'E' ; done ;
             match (depile p) with
             | k when k=j -> print_char 'D' ; aux (max i j) q
             | _ -> false
     in aux 0 ;;
   ```
4. Montrer enfin que toute permutation peut être engendrée à l'aide de deux piles, et rédiger la fonction Caml correspondante.
   La fonction précédente ne permet pas d'engendrer une permutation lorsqu'au moment de dépiler j, ce dernier ne se trouve pas au sommet de la pile. Dans ce cas, il suffit de stocker temporairement dans une seconde pile les éléments situés au dessus de lui, puis de les faire réintégrer la pile initiale une fois j dépilé.

```
let transfert q p =
   let rec aux () = empile (depile q) p ; print_char 'd' ; aux () in
   try aux () with Empty -> () ;;

let rec cherche j p q = match depile p with
   | k when k = j -> print_char 'D' ; transfert q p
   | k -> empile k q ; print_char 'e' ; cherche j p q ;;

let génération =
   let p = creer() and q = creer() in
   let rec aux i = function
       | [] -> ()
       | j::r -> for k = i+1 to j do empile k p ; print_char 'E' ; done ;
                 cherche j p q ;
                 aux (max i j) r
   in aux 0 ;;
```

La fonction **cherche** empile dans q les éléments de p qui se trouvent au dessus de j ; une fois trouvé, les éléments de q sont de nouveau remis dans p.
Ces opérations d'empilement et de dépilement accessoires dans la pile q sont codées par les lettres e et d.